C-Mix/II User and Reference Manual

The C-Mix/II team at DIKU

May 12, 2003

> This is the best manual we have for C-Mix/II at present, which is not to say that it could not be much better.
> Text in boxes like this is not manual text but comments for the developers

1 Introducing C-Mix/II

Those of us who have been in the game long enough to have started programming in interpreted BASIC on a machine with an astronomical 48KB of RAM remember the shocking relevation that for Real Programming Languages running your program was a two-step process: one had to *compile* one's program before it could run. What an amazingly bizarre arrangement! We quickly learned, however, that the clumsiness paid well: compiled programs are much faster than equivalent interpreted programs, typically by an order of magnitude or more[1].

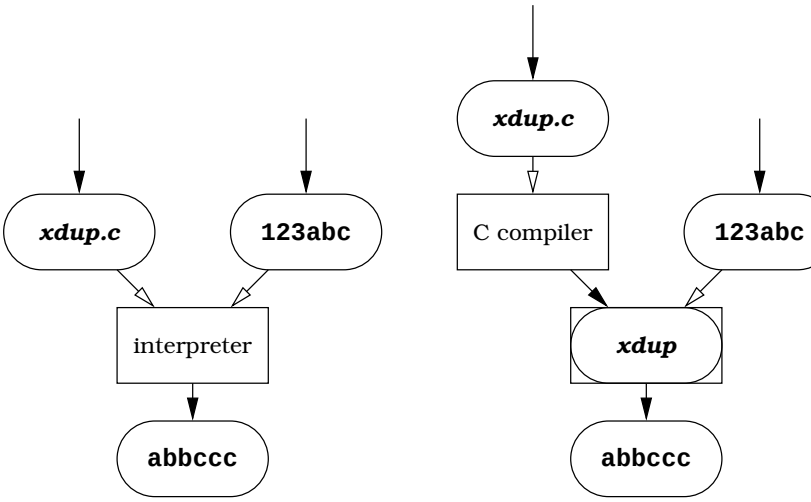
Partial evaluation does something similar to the process of running a program: what used to be a single process is split into two. Somewhat analogous to the compiler, we introduce a utility called a *specializer* (or, as it is also called: a *partial evaluator*) that transforms the program before we run it.

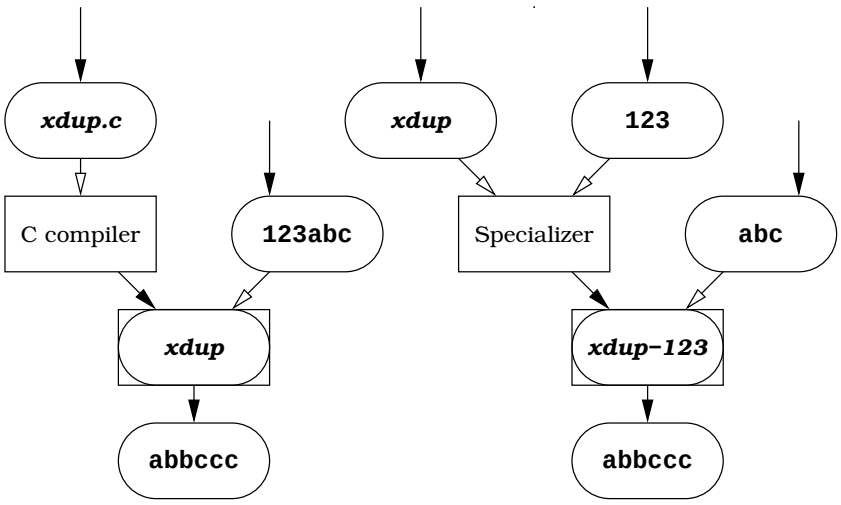
The objective is the same as when compiling: the transformed program often runs much faster than when we run it directly[2]. But the method is different: The compiler speeds up the program by expressing it in a language closer to the hardware of the machine. The specializer needs to be told some of the program's input, and uses that knowledge to reduce the program's workload. This is illustrated in figure 2.

The similarities between compiling and partial evaluation are mentioned in an attempt to ease understanding, not to suggest the specializer as a replacement for the compiler. With C-Mix/II you still compile your program, but you do it after having improved the source code by partial evaluation.

[1] The reader will allow us a little rethorical latitude with respect to the historical facts. The author *is* aware that BASIC might not be an ideal programming language but by any sensible definition certainly is a *real* language; that compilers for BASIC do and did exist; that compilers were used before interpreters were, and so on.

[2] The increase in speed is not always as big as the orders of magnitude yielded by compiling, though in some cases it can be.


xdup.c
123abc
interpreter
abbccc
xdup.c
C compiler
123abc
xdup
abbccc


xdup.c
C compiler
123abc
xdup
abbccc
xdup
123
Specializer
abc
xdup−123
abbccc

1.1 A simple example

The easiest way to describe what partial evaluation does is probably by a concrete example: Consider the program *xdup.c*, alluded to in figures 1 and 2. For the moment the presentation will be easier to follow if we assume the "program" is only a single function whose input `123` and `abc` come in two separate string parameters (later we'll get back to the general case where *xdup* reads its input from *e.g.* `stdin`). Then the source *xdup.c* might look thus:

```
#include <stdio.h>
void xdupfun(char counts[], char data[]) {
    int i, t ;
    for( i=0; counts[i]; i++ ) {
         t = counts[i]-'0' ;
         while( t-- )
             putchar(data[i]) ;
    }
}
```

The specializer, presented with this *subject program*, and the information that `counts` is `123`, sees that the values of the local variables `i` and `t` never depends on the values in the `data` parameter. Thus all of the `i` and `t` manipulation can be done as soon as "`123`" is available—that is, during the specialization process: we say that it is done *at spectime*[3]. Conversely the indexing of the `data` array cannot happen until the array is there.

The specializer produces[4] a file *xdup-123.c* looking like

```
/* This program was generated by C-Mix/II
 *
 * THIS IS AUTOMATICALLY GENERATED CODE
 */
#include <stdio.h>
void
xdupfun(char *residual_arg)
{
  char *data;
  data = residual_arg;
  putchar((int )data[0]);
  putchar((int )data[1]);
  putchar((int )data[1]);
  putchar((int )data[2]);
  putchar((int )data[2]);
  putchar((int )data[2]);
  return ;
}
```

It is reasonable to expect this *residual program* (called so because it is what is left when the specializer has removed everything it can get away with) to run faster than the original *xdup*, because it does not have to spend time with parsing the `counts` parameter.

[3] "Spectime", of course, is a contraction of "specialization time". We use the shorter version because the concept occurs so frequently with C-Mix/II that it was impractical to only have a 6-syllable word for it.

[4] This example is copied verbatim from actual output from C-Mix/II, except that we removed some blank lines.

If you had a program that called

```
xdupfun("123",something);
```

thousands of times, the time needed to specialize *xdup.c* might pay back well in reduced execution times.

1.2 Some advantages of partial evaluation

"This is all very well," the sceptical reader may object here, "but might I not just as well have written *xdup-123.c* manually?" The answer is that yes, in some cases you might. But there are also reasons why you might benefit from the partial evaluation approach:

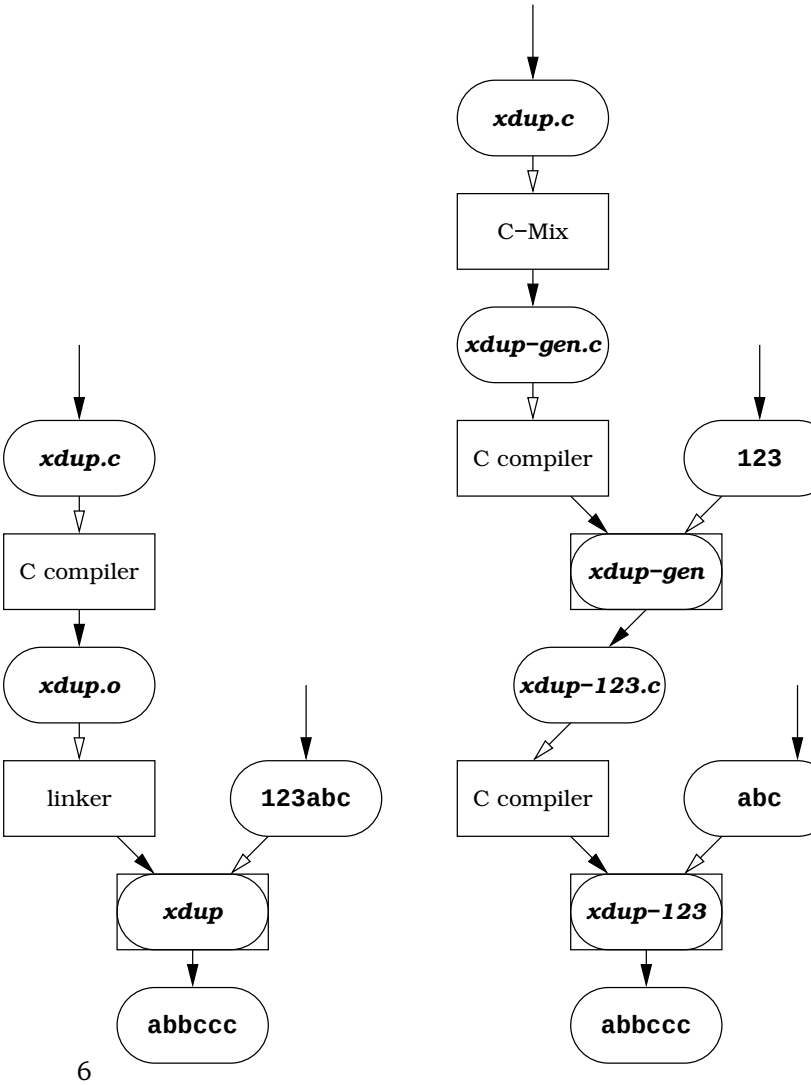
- You can easily change your mind about what you need. If you later find out that what you really want is not *xdup-123.c* but a *xdup-456.c* or *xdup-117.c*, you would be faced with the task of developing a new function from scratch. With C-Mix$\frac{1}{\text{II}}$ that would be as easy as rerunning an automatic utility with some revised parameters. You will save coding time as well as debugging time: even if you do not trust C-Mix$\frac{1}{\text{II}}$ so much as to completely skip testing the automatically generated *xdup-456.c*, C-Mix$\frac{1}{\text{II}}$ certainly makes fewer trivial typos and spelling errors than a human programmer.
- You may have *xdup.c* at hand already—perhaps because you're trying to speed up an existing program, or the code may just be lying around from an early prototype of the program you're developing. Your advantage is the same as when you change your mind.
- It may actually be easier to develop *xdup.c* and specialize it than to develop *xdup-123.c*. This is hardly so in the example—because it was deliberately chosen simple enough that the residual program is easy to understand—but that is not an universal truth; a residual program can easily have a more complex structure than the original subject program.

It might actually be worthwhile to think of the partial evaluator not as a tool for speeding up programs but as a tool that lets you develop efficient customized programs (*xdup-123* etc.) with little more development and maintenance effort than that of a single, general version (*xdup*). Analogously, a compiler can be viewed as either a program that speeds up other programs, or as a utility that saves you the time it would take to write your programs directly in assembler language for each machine type they have to run on.

1.3 How to run C-Mix$\frac{1}{\text{II}}$ on the example

Now we describe what exactly must be typed to specialize the *xdup.c* shown on page 4 with C-Mix$\frac{1}{\text{II}}$. It is a little more involved than it seems in figure 2, because the specialization which appears as a "black box" on figure 2 actually involves several steps when using C-Mix$\frac{1}{\text{II}}$. It will later become apparent that this makes C-Mix$\frac{1}{\text{II}}$ much more flexible, just as enhancing a compiler with a separate linker step makes *that* more flexible.

Figure 3 is closer to the full truth about C-Mix$\frac{1}{\text{II}}$ than figure 2 but a little more complex than we dared present to an unprepared reader.


xdup.c
C compiler
xdup.o
linker
123abc
xdup
abbccc
xdup.c
C−Mix
xdup−gen.c
C compiler
123
xdup−gen
xdup−123.c
C compiler
abc
xdup−123
abbccc

Now, assume that you have C-Mix-II installed on your system and you have typed in *xdup.c* in a working directory. Now, give the command

```
cmix xdup.c -e 'goal: xdupfun($1,?)'
```

Let us have a closer look at that:

`cmix` is the name of C-Mix-II's main program.

`xdup.c` is the C source file that contains the subject program.

`-e` specifies that the next argument is a *specializer directive*. Specializer directives supply C-Mix-II with information that does not directly follow from the subject program alone. Section 5 of this manual describes all of the possibilities; here we just use one.

`goal` is a keyword for this kind of specializer directive.

`xdupfun` is the name of the function that we want to specialize. The input to C-Mix-II may include several functions, so it needs to be told which is the one of which we want to produce a specialized form.

`$1` specifies that the first argument to `xdupfun` will be given at spectime. Note that we do not yet specify what it is; its value will be given later, as the first (hence the `1` in `$1`) argument to the generating extension.

`?` specifies that the second argument to `xdupfun` will not be known at spectime.

If everything goes well, C-Mix-II will output two files: the generating extension *xdup-gen.c*, and an *annotated subject program* in the file *xdup.ann*.

The annotated subject program is an encoded version of the C input with annotations that aim at showing what will happen to the program as it gets specialized. The file is not human-readable; you need to use the `cmixshow` to read it. Give the command

```
cmixshow xdup.ann
```

and you will be shown a pretty-printed version of your program.

Some parts of the program will be shown in boldface by `cmixshow`. Those are the parts of the program that have been annotated *residual*, meaning that they are not executed at spectime but instead appear in the residual program.

Having verified that the annotated program looks okay, the next step is to compile the generating extension. This is straightforward:

```
cc xdup-gen.c -o xdup-gen -lcmix
```

The `-lcmix` links in a C-Mix-II support library for generating extensions. You might need to use additional options for the compiler to locate this library.

Compiling *xdup-gen.c* should produce an executable called *xdup-gen*. Now is the time to present the actual spectime input. Run *xdup-gen*:

```
./xdup-gen 123
```

and it outputs a residual program that is essentially equivalent to the one shown on page 4 though the lay-out may be a little different. If you had needed the residual program as more than an example, you'd probably have redirected the output of *xdup-gen* to a file, e.g. *xdup-123.c*.

The whole process may seem somewhat complicated, but most of it is easily automated. The rest of this manual describes ways to control and customize the individual steps.

1.4 Advanced uses of `cmixshow`

`cmixshow` can do much more than just tell you which parts of the subject program have been annotated residual. It can also show you *why* it is so.

To use the advanced features of `cmixshow` you have to give the option `-s` on the command line to `cmix` (note: `cmix` gets the `-s` switch, not `cmixshow`). The most prominent effect of `-s` is that the annotated program as shown by `cmixshow` gets transformed into spaghetti code with lots of gotos and no structured control statements. This form is closer C-Mix/II's internal representation of the program; it is needed in order to be able to express the more subtle facts about why things become residual.

The true power of `cmixshow` is unleashed if you define the environment variable `BROWSER` to the name of you favorite frames-capable WWW browser before you start `cmixshow`. Then `cmixshow` will spawn a web browser to show you the annotated program, with most of the identifiers hyperlinked to their definitions.

Now you can enable hyperlinks from the program text to explanations produced by C-Mix/II's analyses. At the top of the program a button labeled "Select visible annotations" button. Click it to get to select which class of annotations you want displayed. The different annotations are organised in a tree-like structure; you can collapse and expand branches to your liking. Initially no annotations are displayed; a common choice is to expand the "BTA" branch and then turn off the "Spectime" annotations.

After clicking on the "Ok, show me those!" button, you'll again see your program, but with the residual parts decorated with hyperlinks that lead to additional information about why they became dymanic (shown in the lower frame of the browser).

The other annotation classes document the results of auxiliary analyses. They are mainly useful when debugging C-Mix/II, but can also sometimes be useful to the end-user. Experiment with showing them to see if you can make sense of the information. (If you can, consider contributing a section to this manual!)

2 Tutorial exercises

The following pages contain some exercises that introduce the use of C-Mix/II in partial evaluation. To use the makefiles with the exercises—which is strongly recommended—you need access to the GNU `make` utility (called `gmake` here), to view the annotated source programs you need access to a WWW-browser like Netscape, to run the C interpreter example in Section 2.6 you need access to the GNU `bison` and `flex` parser and lexer generator utilities, and to view ray tracing online you need the SRGP, PTC or OpenGL graphics library [SRGP 1990, PTC 1998, OpenGL 1999].

2.1 Getting started

To avoid confusion, each C-Mix/II example resides in its own subdirectory of the directory `$CMIXDIR/examples`, where `$CMIXDIR` is the directory on your server that C-Mix/II resides in. We suggest you start by making a copy of them in a

working directory, in these exercises we will assume your working directory is called `~/cmix/exercises`. On a UNIX system this can be done by typing something like this (you type in the commands in **boldface**):

```
[~] mkdir -p ~/cmix/exercises
[~] cp -R $CMIXDIR/examples/* ~/cmix/exercises
[~] ls ~/cmix/exercises
ack         cint        matrix      printf
binsearch   fft         pow         turing
[~]
```

To try out your first example, type **cd ~/cmix/exercises/pow** to change to the `pow` directory, and view the source file by typing **cat pow.c**:

```
[~] cd ~/cmix/exercises/pow
[~/cmix/exercises/pow] cat pow.c
double pow(double x, int n)
{
  double a = 1;
  while(n--) a *= x;
  return a;
}
[~/cmix/exercises/pow]
```

The simplest way to process files with C-Mix$_{\text{II}}$ is by using a makefile. Run your first C-Mix$_{\text{II}}$ example by typing **gmake**, producing something like this:

```
[~/cmix/exercises/pow] gmake
gcc -O3   -c pow-time.c -o pow-time.o
gcc -O3   -c pow.c -o pow.o
cmix -s -q -e 'goal: pow(?,$1) producing ("pow_res")' pow.c  (run C-Mix_II to
Info: Partially static structures turned off     make the generating extension)
gcc -I /usr/local/topps/mix/lib/cmix -L /usr/local/topps/mix/lib/cmix
    -o pow-gen pow-gen.c -lcmix              (compile the generating extension)
./pow-gen 15 | (gindent -br -ce || cat) > pow-res.c     (run generating exten-
sion to
gcc -O3   -c pow-res.c -o pow-res.o                  produce residual program)
gcc -o pow-time  pow-time.o pow.o pow-res.o     (link source & residual program
648 pow.o                                                  for timing purposes)
609 pow-res.o
./pow-time 10000000 2 15 >> pow-time.txt
Timing source program...
Timing residual program...
Done.
head -8 pow-time.txt; tail -8 pow-time.txt
============================================================
Program pow.c on host embla, architecture hp700pa2ux10:
Compiler command is `gcc  -O3'
Input is `2 15'

Source   program size: 648 pow.o
Residual program size: 609 pow-res.o

Source   result: 32768.000000
Residual result: 32768.000000
```

```
                  User time       (System time   )
Source   program    3.68 seconds  (     0 seconds)
Residual program    1.33 seconds  (     0 seconds)
Speedup factor       2.8          (   1.0        )
============================================================
rm pow-gen pow-time.o pow-gen.c pow-res.o pow-time pow.o
[~/cmix/exercises/pow]
```

The makefile is set up as explained in the next section to call C-Mix/II for specializing the power function to spectime `n=15`, and then do some timing comparisons: The source program is run 10000000 times with `x=2` and `n=15`, and the residual program is run 10000000 times with `x=2`.

If you find the directory is filling up, you can remove automatically generated files by typing **gmake clean**. This can also be useful if you have adjusted some specialization parameters and you want to make sure everything is recomputed.

Exercise: View the annotated source program that the C-Mix/II analyses produce by typing **cmixshow pow.ann**. If your `$BROWSER` environment variable is set up correctly (e.g. as "`netscape`"), this will start up a browser window, like the one shown in Figure 4. Click on and select which

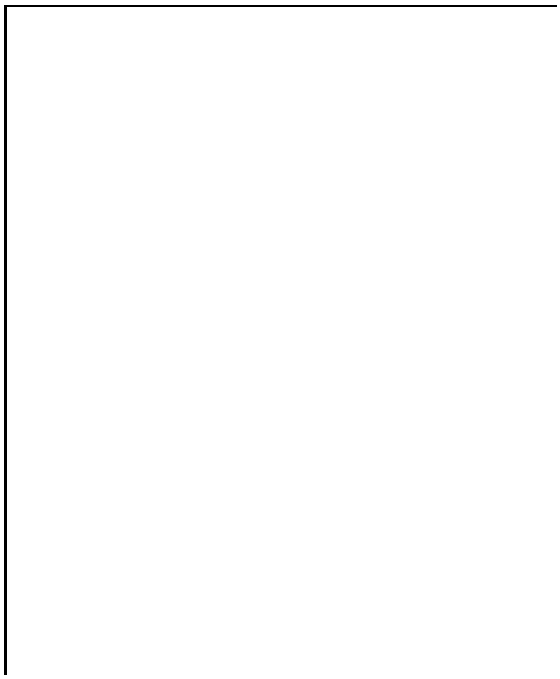


Figure 4: The source code annotation browser

type of annotations, e.g. residual program parts, you would like to see details about. These details will then appear in the lower part of the browser when you

click on a program construct. If you re-specialize by invoking C-Mix$_{II}$ again, you can update the browser window by clicking on the reload button.

Exercise: Print out the source and residual programs (source program: make sure the top frame is selected by clicking somewhere in it, then click on the print button in your browser, residual program: by using your local command for printing ASCII files, e.g. **lpr pow-res.c**) and compare to see what C-Mix$_{II}$ has reduced and unfolded.

2.1.1 Doing it on your own

If you want to make your own example and run it through C-Mix$_{II}$, we suggest you create a new directory with makefiles etc. by using the `pow` example directory as a template.

2.2 Exercise: text formatting with `printf`

Change to the `printf` directory and take a look at the source file `prntf.c`; it contains four functions: a function `mini_printf(fmt, values)` that handles a small subset of the formatting of the standard C library function `printf`, a function `putint(i)` for printing integers, the power function `power(x, n)` and a goal function `goal(x, n)` that acts as an entry point for C-Mix$_{II}$.

The aim is to specialize away all operations on the formatting string `"Power = %d\n"` and the spectime variable `n`. This is set up in the top of the makefile:

```
#---- EXAMPLE SPECIFIC SECTION -------------------------------

SOURCEPROG      = prntf.c
GOALFUN         = goal
BINDTIMES       = (?,$$1)
TESTDATA        = 2 23
SPECTIMEDATA    = 23
REPEAT          = 100000
CFLAGS          = -O3
CMIXFLAGS       = -s -q

#---- END OF EXAMPLE SPECIFIC SECTION ------------------------
```

The binding times of the parameters to the `goal` function are given by "`(?,$$1)`", indicating that the first parameter, `x`, is residual (unknown) and `n` is a spectime parameter given as argument number 1 to the generating extension. The example is specialized with spectime data $\texttt{n} = 23$.

Try making the residual program by typing **gmake prntf-res.c** and take a look at it—notice how the format string is now built into the specialized `mini_printf` function.

The program `prntf-time.c` performs timing and comparisons of the source and residual programs. The first few lines look somewhat like this:

```
/**** EXAMPLE SPECIFIC SECTION ***************************/

#define ARGS 2
```

```
extern int goal(int x, int n);
extern int goal_res(int x);
int x; int n;
#define READARGS     x = atoi(argv[1]); n = atoi(argv[2])
#define CALLSOURCE   goal  (x, n)
#define CALLRESID    goal_res(x)
#define FORMATRESULT "%d"

/**** END OF EXAMPLE SPECIFIC SECTION ********************/
```

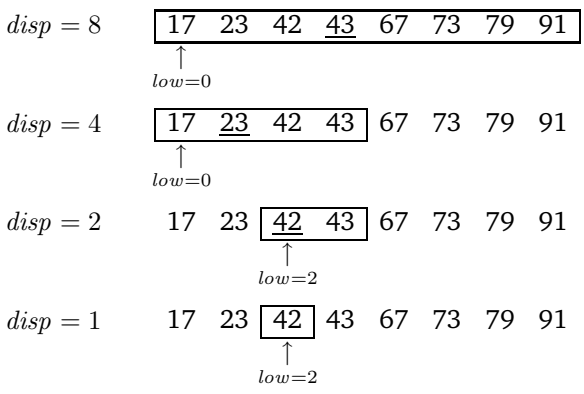
The C macro `ARGS` indicates how many arguments the source goal function `goal` takes. This function is declared, along with the specialized `goal_res` function. C Macro `READARGS` reads and converts the arguments from the command line in usual C style, while `CALLSOURCE` and `CALLRESID` perform the calls to the source and residual functions. The result of each of these functions is formatted using the `FORMATRESULT` string.

When the timing program `prntf-time` is run via the makefile, the `TESTDATA` input from the makefile is passed, along with the `REPEAT` value which indicates how many times the calls should be repeated. This is to make sure that the running times are not too short for timing purposes.

Exercise: Try making alterations and additions to the `prntf.c` program or the top of the makefile, and then do **gmake timing** after each: what happens if you add several calls to `mini_printf`? With different spectime arguments? With identical spectime arguments? Try also getting an estimate of the speedup by typing **gmake timing**—speedups around 1 to 8 are usual for partial evaluation.

2.3 Exercise: Binary search

Given a sorted array of numbers, finding the index of some given number a can be done in logarithmic time by comparing a with an array element near the middle and continuing in this fashion with either the upper or lower half of the array. Looking for 42 in the list $17, 23, 42, 43, 67, 73, 79, 91$ can be done in 4 steps:



At each step, a is compared with the underlined element, the displacement *disp* is halved, and the offset *low* is adjusted accordingly.

A function `binsearch(x, size, a)` that searches for `x` in an array `a` of length `size` initialized to contain $[0, 1, 2, ..., \texttt{size} - 1]$ can be found in the file `bsrch1.c` in the `binsearch` directory. Try specializing and timing this function with spectime `size` $= 512$ by typing **gmake timing1** (this may take a while).

As you can see, the specialization has produced a speedup of around 4, but at the cost of a residual program about 35 times the size of the source program! Take a look at the residual program (`bsrch1-res.c`). Before you continue reading, consider the following question: what makes it so huge? (Hint: what happens when you change the spectime value 512?)

The problem is that the residual program contains one `if` statement for each element in the array—a total of 512 in this case! This kind of code explosion is often undesirable, and is a result of "too much" specialization. This can be improved by instructing C-Mix/II to residualize some constructs that actually could be performed at spectime. Given a function with a variable `y` that should be residualized:

```
int f(int x) {
  int y;
  ...
```

this is done by adding a `#pragma` to the source file like this:

```
int f(int x) {
#pragma cmix residual: f::y
  int y;
  ...
```

(This can also be specified in a separate `.cmx` file, cf. the User Manual.)

Exercise: Make a copy of `bsrch1.c`, call it `bsrch2.c`, and take a look at the annotations by typing **cmixshow bsrch1.ann**. Decide which variable, `disp` or `low`, should be residualized, insert the appropriate `#pragma` in the `bsrch2.c` file, and examine the differences in code, speedup and code explosion by typing **gmake timing1 timing2**. (A correct solution can be found in `answr2.c`).

2.4 Exercise: Ackermann's function

In directory `ack` you will find a small program in `ack1.c` for computing Ackermann's function:

$$Ack(m,n) = \begin{cases} n+1, & \text{if } m = 0 \\ Ack(m-1,1), & \text{if } m > 0 \wedge n = 0 \\ Ack(m-1, Ack(m, n-1)), & \text{if } m > 0 \wedge n > 0 \end{cases}$$

Try specializing Ackermann's function for spectime $m = 3$ by typing **gmake timing1**. The resulting program, `ack1-res.c` is a little messy because it is machine-generated, but a tidy version of it can be found in `ack1-tidy-res.c`.

Note that there are three calls to functions where the arguments are the constant 1 that have not been reduced. The reason is that C-Mix/II makes a *monovariant* binding-time division: in all calls to `ack`, the second parameter must

be residual (because it is so in *some* calls). Make a copy of file `ack1.c`, call it `ack2.c`, and figure out a way to circumvent this problem by hand-rewriting the program (a solution is given in `answr2.c`). Specialize it by typing **gmake timing2**. A tidy version of the residual program can be found in `ack2-tidy-res.c`.

2.5 Exercise: Turing machine

In the `turing` directory you will find a small Turing machine interpreter in file `turng1.c`. The machine has an instruction pointer i, a tape pointer p and takes a list of instructions. Initially, p points to the first element of the tape, and the Turing machine handles these instructions:

Left	move p one step left on the tape, unless $p = 0$
Right	move p one step right on the tape, unless $p = tapemax$
Write a	write character a at the current tape position
Goto i	let next instruction be instruction i
If a goto i	if the character at the current tape position is equal to a, let the next instruction be i
Stop	stop and return the part of the tape that starts at the current tape position

The following Turing program that can be found in file `first0to1.t` finds the first '0' on the tape, changes it into a '1', and returns the rest of the tape, starting at this '1':

0 *If* '0' *goto* 3
1 *Right*
2 *Goto* 0
3 *Write* '1'
4 *Stop*

For instance, running this program on tape 110101 returns 1101.

The function called `loadAndRunTuring(char* filename, char tape[], int tapelen)` loads a Turing program from a file and then calls another function `turing(char tape[], int tapelen)` that implements the Turing machine.

Beginner's exercise: Specialize the turing interpreter in file `answr1.c` by typing **gmake answer1**, then view the residual source program `answr1-res.c` to see what can be produced.

Now try specializing the turing interpreter in file `turng1.c` by typing **gmake**. Finally, look at the function `turing(char tape[], int tapelen)` in both source files and explain the different specialization results.

Advanced exercise: Try specializing the program with spectime instruction list `instruction`, instruction pointer `i`, tape pointer `p`, and residual `tape` by typing **gmake**. What is the problem? Look at the annotated program by typing **cmixshow turng1.ann**; what is the reason for this problem? Solve it by inserting appropriate `#pragmas` (a correct solution can be found in `answr1.c`).

The problem is *infinite specialization:* C-Mix II does not know the length of the tape at specialization time, so it generates specialized code for all possible

positions of the tape pointer—of which there are infinitely many. By forcing the tape pointer to become residual, this infinite specialization is avoided.

Take a look at the residual program; note how well it resembles the Turing program! Only now it is in C code, so we have in effect obtained a compilation of Turing code into C code!

Slightly time-consuming exercise: code blowup What happens if you specialize with `tapelen` as a spectime constant of 500? This requires alterations in the makefile so that it reads

```
#---- EXAMPLE SPECIFIC SECTION -------------------------------

SOURCEPROG      = turng.c
GOALFUN         = loadAndRunTuring
BINDTIMES       = ($$1,?,$$2)
TESTDATA-turng1 = first0to1.t "110101---" 500
SPECDATA-turng1 = first0to1.t 500
REPEAT          = 1000000
CFLAGS          = -O3
CMIXFLAGS       = -s -q
TESTDATA-turng2 = 3right1left.t "001000110---"
SPECDATA-turng2 = 3right1left.t
TESTDATA-answr1= $(TESTDATA-turng1)
SPECDATA-answr1= $(SPECDATA-turng1)
TESTDATA-answr2= $(TESTDATA-turng2)
SPECDATA-answr2= $(SPECDATA-turng2)

#---- END OF EXAMPLE SPECIFIC SECTION ------------------------
```

and file `turng-time.c` to read

```
/**** EXAMPLE SPECIFIC SECTION ***************************/

#include "turng.h"
#define ARGS 3
extern char* turing(char[], int);
extern char* turing_res(char[], int);
char* tape; int tapelen;
#define READARGS     readTuringProgram(argv[1]); \
                     tapelen = atoi(argv[3]); \
                     tape = (char*)malloc(2 * tapelen * sizeof(char)+1);
#define CALLSOURCE   (strcpy(tape, argv[2]), turing    (tape, tapelen))
#define CALLRESID    (strcpy(tape, argv[2]), turing_res(tape, tapelen))
#define FORMATRESULT "%s"

/**** END OF EXAMPLE SPECIFIC SECTION *********************/
```

(Remember to remove any `residual: turing::p` directive.) What are the speedup and code blowup factors like? What seems to be the best binding time for tape pointer `p`?

Exciting, time-consuming exercise!

Make a copy of `turng1.c`, calling it `turng2.c` and extend the Turing machine by adding a new instruction *ReadGoto* which reads a digit i off the tape and jumps to that instruction—a "computed goto." (A solution can be found in `answr2.c`.) The following Turing program, found in file `3right1left.t`, uses the *ReadGoto* instruction to repeatedly go 3 steps right or 1 step left according to whether the current symbol is '1' or '0', replacing any '1's by '0' as it does so. When two '1's in a row are encountered, it goes 3 steps left and stops.

0	*Goto* 6	7	*Left*
1	*Right*	8	*ReadGoto*
2	*Right*	9	*Left*
3	*Right*	10	*Left*
4	*If* '1' *goto* 9	11	*Left*
5	*ReadGoto*	12	*Stop*
6	*Write* '1'		

For instance, running this program on tape 001000110 returns 111110. The modified interpreter can be specialized to this program by typing **gmake timing2**—remember to set up the makefile and timing program as they were originally if you have changed them in the previous exercise.

Can you still specialize the Turing interpreter with spectime instruction list?

The present problem arises often in interpreters: the index of the next (spectime) instruction depends on some residual data, yet we *know* that it can only be one of finitely many different values. Can you think of a way of exploiting this *bounded static variation* so that the instruction list can be specialized away?

The solution is so often seen in partial evaluation that it has been named *The Trick*: when an assignment `s = d` of a residual value `d` to a spectime variable `s` that can only evaluate to finitely many different values must be performed, a switch statement is inserted in the source program:

```
switch (d) {
  0 : s = 0; break;
  1 : s = 1; break;
  ⋮
  n : s = n; break;
}
```

2.6 Exercise: C interpreter

In directory `cint` you will find a file `cint1` containing an interpreter for a small subset of the C language: the interpreted language includes functions, integer variables and constants, some arithmetic operators, if- and while-statements.

This is an example of supplying some of the spectime input not on the command line but in a text file. This requires parsing, which is performed by a parser generator generated parser that is not specialized—instead it is linked into the generating extension as a spectime function `readProg(char* filename)`.

Try specializing the interpreter to the power function defined in `power.c` by typing **gmake timing1**. Note the speedup factor, which is due to more or

less removing a complete layer of interpretation: all handling of the syntax of `power.c` has disappeared from the residual program.

The residual program contains rather a lot of small functions; a slightly tidier version can be found in `cint1-tidy-res.c`. Try comparing programs: are there any similarities between `power.c` and `cint1-tidy-res.c`? Between `cint1.c` and `cint1-tidy-res.c`?

The evaluation of if-statements is missing. Make a copy of `cint1.c`, call it `cint2.c`, add the missing code, and specialize the interpreter to the greatest common divisor function in `gcd.c` by typing **gmake timing2** (a solution can be found in `answr2.c`).

Interesting but probably very time-consuming exercise! Currently, the interpreter can handle several function *definitions*, but not function *calls*. Try extending the interpreter, e.g. with function calls. Make sure that all syntax handling is still done at specialization time.

2.7 Exercise: Matrix operations

In the `matrix` directory you will find a program `mtrix1.c` that contains the following matrix and vector operations for $M = 25$:

`vector solve(matrix A, vector b);`	Solve $Ax = b$ and return x for $M \times M$ matrix A and $M \times 1$ vectors x and b
`vector mulMatVec(matrix A, vector b);`	return Ab for matrix A and vector b
`char* printMatrix(matrix A);`	return a text representation of A
`char* printVector(vector b);`	similar to `printMatrix`
`matrix makeMatrix(unsigned int seed);`	return a matrix with pseudo-random numbers deterministically generated from *seed*
`vector makeVector(unsigned int seed);`	similar to `makeMatrix`

At the bottom of the file you will find the goal function:

```
vector goal(unsigned int Aseed, int n) {
  int i;
  matrix A;
  vector x, b;
  i = 1;
  A = makeMatrix(Aseed);
  while (i < n) {
    x = makeVector(i * 523 % RANDMAX);
    b = mulMatVec(A, x);
    printstatus("/* solving A * x(i) = b for i = %d */\n", i);
    solve(A, b);
    i++;
  }
  x = makeVector(i * 523 % RANDMAX);
  b = mulMatVec(A, x);
  return solve(A, b);
}
```

Using `Aseed` to generate a pseudo-random matrix `A`, it generates `n` different `b` vectors and solves `Ax` = `b`. Try specializing this goal function with residual `n` and spectime `Aseed` = 17 by typing **gmake timing1**. What is the problem? Look at the annotated program by typing **cmixshow mtrix1.ann**; before reading on, consider what the reason for this problem is?

The problem here is *infinite specialization:* the counter `i` in the while loop is spectime, but the loop is controlled by the *residual* limit `n`. As C-Mix$_{II}$ doesn't know the value of `n` yet, it tries to specialize for `i`=1,2,3,..., never terminating.

Solve the problem by inserting a `#pragma` (a correct solution can be seen in `answr1.c`). Find the speedup and code blowup for the solver with matrix `A` as a spectime constant by typing **gmake timing1**.

Interesting, time-consuming exercise: Make a copy of the file `mtrix1.c`, call it `mtrix2.c`, and make some of your own matrix and vector operations, e.g. dot product, matrix multiplication and addition, multiplication with a constant, etc. Try specializing an expression like $cA + b \cdot d$ where c is a spectime simple constant, A a residual matrix, b a spectime vector constant, and d a residual vector, by typing **gmake timing2**. What are the speedups like?

It can be desirable to do algebraic reductions like $0A = \mathbf{0}$, $1A = A$, $\mathbf{0} \cdot b = \mathbf{0}$, $\mathbf{1} \cdot b = b$. This can be done by inserting tests like `if (c == 1) return A;` if `c` is a spectime constant, matrix `A` is not rebuilt by a lot of additions and multiplications, but simply returned unchanged, and the test disappears from the residual program! This technique is called *partial evaluation with selective simplification*. What are the speedups for $cA + b \cdot d$ like when c or b are 0 or 1?

2.8 Exercise: FFT

The FFT (Fast Fourier Transform) algorithm implements a discrete Fourier transformation, and has been subject for numerous hand-optimizations. Given an array of data, FFT first "scrambles" the data, which enables a second phase to use a divide-and-conquer algorithm to compute the resulting array. The first phase only moves data around, whereas the second phase does actual computations, including multiplications, additions and calls to math library functions.

Exercise: In the `fft` directory you will find a file `fft1.c` containing a generic implementation of FFT. Input is copied from a set of external arrays, `orig_real` and `orig_imag`, which are initialized by a `generate_data` function that takes a seed `q` to generate some test data. The size of the arrays is defined in file `fft.h`. Try specializing `fft1.c` by typing **gmake**; how big is the speedup and code blowup?. Compare the source file with the residual file `fft1-res.c`: what seems to be the main source of speedup for the FFT?

Given that the `fft` function operates on an array of values, it must perform a fair amount of array indexing, but as the size of the array is fixed, C-Mix$_{II}$ can actually do this indexing at specialization time!

Beginner's exercise: Try specializing the FFT in file `answr2.c` by typing **gmake answer2**. Compare residual programs `fft1-res.c` and `answr2-res.c`: how does specialization time indexing manifest itself? Does it have an effect on the speedup? Also, take a look at the `real` and `imag` array definitions in the annotated source program by typing **cmixshow answr2.ann**.

Advanced exercise: Make a copy of file `fft1.c`, call it `fft2.c`, and change it to achieve spectime indexing of the `real` and `imag` arrays (a solution can be found in `answr2.c`). Try it out by typing **gmake timing2**; compare residual code, speedups and code blowups with the previous specialization.

Specializing away array indexing into arrays containing residual data is achieved by *array splitting*, in which the residual code contains individual variables for the elements of the array. For instance, a *spectime* array of *residual* values, `int d[3]`, will result in residual variables `int d0, d1, d2`. The choice as to whether an array of residual data is split or not depends on the *pointers* used to point to the array elements: the array will be split exactly when they are *spectime* pointers to residual data. Note that during the internal transformation in C-Mix$_{II}$ from C to Core C plain integer array indexing like `a[i + 42]` is converted into pointer manipulation.

In the present example, we split the array by removing the `#pragma` that forces the second and third parameters of `fft(int is, REAL *fr, REAL *fi)` to become residual pointers.

2.9 Exercise: Ray tracer

A *ray tracer* is a program for producing images of scenes that consist of various objects, viewed from a specific direction, illuminated with some light sources etc. Each object, view direction, light source etc. are described by coordinates in a euclidian space, and the ray tracer then traces all the light rays backwards from the viewer's eyepoint to the objects in the scene, changing direction if it hits a reflecting object.

In the `ray` directory you will find a simple ray tracer capable of tracing images containing squares, discs and spheres with uniform or checked surfaces. It consists of the following files:

`ray.c`	The main tracer functions to be specialized
`scenes.c`	A number of scenes to trace
`main.c`	The user interface
`create.c`	Utility functions for creating objects in the scene
`vector.c`	Utility vector functions
`mypower.c`	The power function
`plotOpenGL.c`	Utility functions for producing screen graphics using OpenGL
`plotsrgp.c`	Utility functions for producing screen graphics using SRGP
`plotptc.c`	Utility functions for producing screen graphics using PTC
`tiff.c`	Utility functions for saving images to TIFF files
`lowerarity.c`	A mediator between `main.c` and the residual goal function
`ray1.cmx`	Specializer directives for C-Mix$_{II}$

The goal function `draw_scene(width, height, scenenum, color)` is located in `ray.c`. It draws one of the scenes found in `scenes.c`.

The entire ray tracer can be built by typing **gmake ray**.

Beginner's exercise: Try specializing the ray tracer with respect to the first scene by typing **gmake answer1**; this will use the specializer directives in `answr1.cmx` to specialize `ray.c`. Now compare `answr1.cmx` and `ray1.cmx` and explain why the differences are necessary to avoid infinite specialization.

Advanced exercise: Specializing `draw_scene` right away by typing **gmake timing1** turns out to produce infinite specialization. Take a look at the `draw_scene` function: find the cause of this problem and remedy it by inserting an appropriate directive in `ray1.cmx` (an answer can be found in `answr1.cmx`).

Note that on most systems, sending the image data to the computer display constitutes a bottleneck, so it is hard to get a realistic picture of the speedup by showing the tracing process simultaneously in two windows. It is better to time first the source program and then the residual program, or to time both simultaneously without sending any data to the display.

Specializing the ray tracer can quickly produce very large residual code for scenes with many objects. To avoid excessive code blowup it can be necessary to force some variables to be residual.

Hard exercise: avoiding code blowup Make a copy of `ray1.cmx`, call it `ray2.cmx` Try specializing the ray tracer with respect to scene 5 by typing **gmake timing2**. While you are waiting for the generating extension to finish, open another window and look at the memory usage (e.g. by running the `top` program). When you are convinced that the specialization will not terminate before the memory is exhausted, abort the specialization.

Now look at the residual program, find the source of code blowup and insert directives in `ray2.cmx` to force the appropriate variable(s) to be residual. Of course you should not make too much residual, or you will sacrifice the speedup! (An answer can be found in `answr2.cmx`.)

3 Running C-Mix/II

The command line syntax for C-Mix/II is quite simple. Most of the turns and knobs that control C-Mix/II's operation are *specializer directives* which have their own section (5) of the manual.

3.1 The C-Mix/II main system

`cmix` *option... filename...*

C-Mix/II accepts the following command line options:

`-I`***path*** – specifies where to look for `#included` files. Equivalent to the `-I` option usually found in C compilers.

`-D`***symbol*** or `-D`*symbol=expansion* – defines the given symbol as a preprocessor macro when reading in the C sources. Equivalent to the `define:` specializer directive and to the `-D` option usually found in C compilers.

`-e`***directive*** – give an arbitrary specializer directive (see section 5). Note that the shell will probably need the directive text to be quoted except in very simple cases.

`-o`***basename*** – controls the naming of the C-Mix/II output files, which will be called *basename-gen.c* and *basname.ann*. Equivalent to the `outputbase:` specializer directive.

`-q` – suppresses routine messages about reading and writing files.

`-v` – be verbose about the progress of analysis phases.

`-s` – output the `.ann` file in a format that resembles the data format uses internally by C-Mix/II. The format is sufficiently close to C to be readable, but you may have trouble recognizing the structure of your program—or indeed to find your program amidst the definitions read from header files.

`-S` – same as `-s` but turns off some shortcuts normally taken to keep the output readable. This makes the output quite unreadable indeed: what normally is represented as `t2p->t1.x` now gets rendered `*&(&(*&t2p)->t1)->x`. This is primaily useful when debugging C-Mix/II.

`-b` – make a generating extension that tries very hard to share code internally in each residual function. This slows down the specialization process very noticeably, usually with no great benefit.

`-E` – instead of parsing the C source and making a generating extension, just preprocess it (with the same include file search path and predefined symbols as C-Mix/II would normally use) and write the preprocessed source to the standard output. This is sometimes useful for debugging if one suspects the program text is changed unexpectedly in the preprocessing step.

`-h` – displays an option summary

`-d`**...** – controls debugging output, which you normally don't want to see.

Any command line argument to C-Mix/II that is not an option is treated as a file name. If the argument ends with `.c` then it is assumed to be a C source file containing code to be specialized.

If you want a file whose name does not have a `.c` suffix to be parsed as C code, you need to use an explicit `source:` directive. E.g.:

```
cmix -e 'source: plain-file-name'
```

If a command line argument is not an option and does not look like a C source file name, it is interpreted as the name of a *script file*. The script file consists of specializer directives separated by semicolons or blank lines. Since most of C-Mix/II's actions can be controlled by specializer directives, once you have set up a script file, a useful C-Mix/II command line can be as short as

```
cmix myproject.cmx
```

C-Mix/II names its output files by appending `-gen.c` and `.ann` to a basename obtained in the following way:

- An `-o` option, if it exists, always takes precedence
- If no `-o` option is found C-Mix II looks for an `outputbase:` specializer directive.
- The name of the first script file named on the command line. By convention, script files have names ending in `.cmx`; this suffix is stripped from the file name before using it to derive output file names.
- If there is no script file, the name of the first C source file named on the command line is used.
- If everything else fails, the output files will be names `cmix-gen.c` and `cmix.ann`.

3.2 The annotated-program browser

The `cmixshow` program can be used to display the annotated program output that `cmix` writes to the `.ann` file. It is invoked thus

`cmixshow` [`-b`|`-u`|`-l`|`-r`|`-h`|`-s`] *filename.ann*

`cmixshow` can display the binding-time annotated programs output by `cmix` for user feedback in various formats, as selected by the options.

`-b` – Display residual parts of the program in bold

`-u` – Display residual parts of the program underlined

`-l` – Display residual parts of the program in bold and spectime parts underlined. This works well for terminals that differentiate between bold and underline.

`-r` – The opposite of `-l`; primarily useful with terminals that use reverse video to emulate underlining.

`-w` – Output HTML that uses bold red text for residual parts of the program and italic green text for spectime parts.

`-s` – Makes `cmixshow` install itself as a local HTTP server and start a WWW browser for interactive inspection of the annotations. This is the only way to inspect the extra annotations enabled by the `-s` or `-S` options to `cmix`.

The `BROWSER` environment variable selects which command to use to start the browser. If the variable is not set, `cmixshow` simply outputs an URL and waits for the user to start a browser.

Each time the browser reloads the top frame, `cmixshow` checks if the `.ann` file has changed and reloads it if appropriate. In this way it is possible to view the new annotations each time `cmix` is rerun just by clicking the "Reload" button in the browser.

If no option is given, `-s` is assumed if the `BROWSER` variable is set, `-b` otherwise.

With the `-s` option, the output goes via a network connection to the browser. In any other mode, if the standard output is a terminal, the pretty-printed output is piped to a pager. Otherwise it is sent to the standard output. Except with `-w`, bold and underlined text is created by backspace overstrike which is understood by most pagers.

3.3 Compiling the generating extension

```
cc -lcmix basename-gen.c -o binaryname
```

For the generating extension to work, you need to link it to a special C-Mix/II support libary. You might need to use an `-L` switch to locate it if it has been installed in a nonstandard place.

3.4 Running the generating extension

If you use the `goal` directive, you can simply run the compiled generating extension, giving as many command-line parameters as there were "$\$n$" specificers in the `goal` directive. It then output specialized C code to standard output.

You can put a "`-R`" switch in front of those parameters. This disables a restructuring phase that tries to express the control flow of the specialized C code with high-level control constructs. Sometimes this makes compilers generate better code for the residual program; sometimes it has the opposite effect. The `-R` switch can also be use if you find the residual program provokes an internal error in the restructuring phase.

If you use `generator` directives, you have to supply a `main` function yourself, as detailed in the description of that directive. Your `main()` can disable the restructuring phase by doing

```
extern int cmixRestruct ;
cmixRestruct = 0 ;
```

4 User annotations: telling C-Mix/II what it can't figure out itself

The core of the C-Mix/II system is a set of automatic analyses that decide which of the actions and values in the subject program should be spectime and which should be residual. Most of the time the analyses do a good job, but they don't perform micracles, nor do they read minds.

User annotations are about telling C-Mix/II that you know better than it which binding time a value or an action should have. Few programs need no user annotations, most programs need a few, and some programs need a lot of them to specialize satisfactorily. This section describes the types of user annotations you can attach to your program and how to do it.

Whether user annotations should be placed in the subject program itself or in a separate file is a matter of style. C-Mix/II supports both choices by expressing (most) user annotations as specializer directives which can appear either in the subject program or in a script file.

4.1 Sorts of user annotations for variables

You can attach these annotations to variables in the subject program:

`spectime` – Use this for variables that you want to exist only at spectime. This doesn't actually change the binding-time analyses, but if C-Mix/II finds any

reason to want the variable to be residual, it will not produce a generator but instead give an error message and terminate.

It can be helpful to place this annotation on some of the variables you hope to specialize away. You will then get immediate response if this is not possible, and the error message might be more helpful in pointing out the problem than the machine-annotated program.

This annotation can not be used for variables that have only `extern` declarations in the C-Mix/II input files. Such variables are always forced to be residual. (But see the `dangerous spectime` annotation below).

`dangerous spectime` – Prevents an external variable with no definition from being residualized. This can sometimes help making the specialization process use less resources, but also has a real potential for making the residual program incorrect, because C-Mix/II cannot track changes to the value of the variable without having control over its definition. *This severely compromises* C-Mix/II*'s ability to produce correct residual programs! Do think the case over at least twice before using this annotation.*

You can safely ignore this annotation until you are sure that you need to do something radical about the efficiency of the generating extension.

`residual` – Use this to tell C-Mix/II to assign binding-times so that the variable becomes completely residual. This normally forces other variables and values to be residual as well; that happens automatically and you need not provide user annotations for those. (Also, it might make it impossible to fulfil another `spectime` annotation).

This annotation is the primary tool for controlling a specialization that would otherwise proceed infinitely or produce unacceptably large residual programs. Examples of the use of `residual` are given in the C-Mix/II tutorial.

`visible spectime` – This has the same effects as `spectime` and additionally specifies that the variable should have external linkage in the generating extension. Namely, the variable will be "visible" to other program modules.

This annotation should be used for variables that you want to be shared between the generating extension and code that you (unbeknownst to C-Mix/II) link together with it. It might either be variables that are initialized by a user-supplied `main` function before the generator functions are called, or variables that are used by external functions annotated as `spectime`.

`visible residual` – This has the same effects as `residual` and additionally specifies that the variable should have external linkage in the residual program.

This annotation is useful for variables that should shared among the residual program and program parts that have not been partially evaluated.

The two `visible ...` annotations can only be applied to variables that have

- external linkage, *and*

- a definition (*i.e.* a file-level declaration without `extern`, but possibly with an initializer)

in the original program. These are the only variables where visibility is a question at all. If you wonder why one would want one of these variables to *not* have external linkage in *e.g.* the residual program, remember that C-Mix/II can specialize several source files together to one, and that the variable might be used only by those files. This is actually what C-Mix/II assumes if you do not tell it otherwise—even if there is only one file: many C programmers are lazy and define their global variables as `int frob;` where `static int frob;` would have been more appropriate.

Note that the order of the words matter: there is no annotation called `residual visible`. The two concepts of "residual" and "visible" are not independent; it would be pointless to just declare a variable `visible` but leave it to C-Mix/II to decide if it should be visible in the generating extension or the residual program!

Objects with external linkage but without a definition are always "visible"; if you do not annotate them with `dangerous spectime`, C-Mix/II will default to `residual`. All other variables without user annotations will have a binding time selected by C-Mix/II and will be internal to the generated source files.

4.2 How to write user annotations for variables

Write a specializer directive reading *e.g.*

```
visible spectime: variable_name
```

For global variables—whether their linkage is internal or external—simply name them in the specializer directive. Local variables (including `static` ones) and function parameters use the two-part syntax (think C++)

annotation-name: *function*::*localname*

Thus,

```
#pragma cmix residual: foo zugol::br
```

in a C source file forces the global `foo` and the local or parameter `br` in function `zugol()` to be residual.

If you try to annotate a global which exists (with internal linkage) in several different files, or a local variable whose name is not unique within the function (because it is defined in several blocks inside it) you will have a warning thrown at you and every matching variable gets annotated. Please contact us if this is a real problem for you.

4.3 Sorts of user annotations for functions

The specialization of functions defined in the subject program (*i.e.* functions whose body is seen by C-Mix/II) can be controlled by adding annotations to their formal parameters or local variables. These functions (even those that have external linkage in the source files) are never "visible" outside either the generating extension or the residual program. (If you think you need to make a function visible in the generating extension, the `generator:` specializer directive described in 5.2 is probably what you are really looking for).

However, when you call a function whose definition C-Mix/II cannot see, C-Mix/II has no way of deducing whether the call should be made at spectime or in the residual program, or what types of side effects may result from the call. A number of annotations are available to control how external calls are handled. They fall in two groups. The first group control which assumptions C-Mix/II makes about possible side effects through the function calls. The second group provides control of when the function call is actually made.

`pure` – The external function does not commit any side effects at all. Its return value does not depend on anything but the argument values (and, if one of the arguments is a pointer, the data that pointer point to, and so on recursively).

Typical functions that fit this description are `sin()` and other mathematical functions[5].

`stateless` – The external function may commit some side effects, but they only depend on the argument values (and the objects pointed to by the argument values, and so on). In particular, the only objects that may have their values changed by the call are those pointed to (perhaps indirectly) by the argument—the function does not have any internal "state".

`rostate` – The external function may still not change any objects not reachable from the arguments, but the values that are written into them and returned from the function may depend on some kind of external state.

`rwstate` – The function can be expected to do anyting a C function may legally do, including changing the values of any objects that it knows the address of, and including changing an external state which affects later calls to `rostate` or `rwstate` functions.

This is assumed as the default by C-Mix/II when no explicit annotation is given.

C-Mix/II only allows `rwstate` calls to happen at spectime if it can prove that they will be performed in precisely the same sequence as if the source program was run unspecialized. Evidently, if the sequence and number of `spectime` calls depend on the program's dynamic input (or if C-Mix/II just isn't smart enough to realize they don't) this is impossible, and an error message will result.

`spectime` – This annotation specifies that all calls of the function should happen at spectime. If C-Mix/II cannot make sure that all parameters in a call are known at spectime, it aborts with an error message which (hopefully) helps you understand why it is so.

This annotation is intended to be placed on *e.g.* functions that read in parts of the spectime input that are not given as arguments to the goal function.

[5] C-Mix/II will automatically insert `pure` annotations for the functions in `math.h` when you include the header file. Actually some of these functions sometimes have the side effect of setting `errno`, *e.g.* when one tries to execute `sqrt(-1.0)`. However, we haven't yet figured out how to make C-Mix/II take this into consideration and still reach a speed benefit. Since nobody ever seems to actually check `errno` after calling one of these functions, we are quietly going to ignore this matter henceforth.

`residual` – This annotation specifies that the function may only be called in the residual program.

`anytime` – This annotation is the default and specifies that the function may be called either at spectime or at residual time, depending on how soon all of the arguments are available.

However, if the function is also annotated `rwstate` or `rostate`, it will only be called at spectime if that is explicitly requested. (That is, for such function `anytime` is really equivalent to `residual`).

4.4 How to write user annotations for functions

This is even simpler than for variables:

```
stateless spectime: gizmo() gadget()
```

You don't need the `::` syntax, or worry about name clashes, since function annotations are only concerned with symbols with external linkage.

In a single annotation you can give either one of `pure`, `stateless`, `rostate`, or `rwstate`, or one of `spectime`, `anytime`, or `residual`, or one annotation from each group.

But this is not the whole story: you can also annotate an individual function call, thereby overriding the (perhaps default) annotation on the function *per se*. This can't be done as a specialization directive, primarily because the developers coundn't think of a good way to specify precisely which call the annotation is aimed at.

Overriding of single function calls is meant primarily for basic I/O functions such as `fopen()`, `fgets()`, etc. that are used to read spectime as well as residual input in the same program. Those calls that read the spectime input can be annotated `spectime`, leaving the reads of residual inputs residual.

You annotate a single function call by prefixing it with `__CMIX(`*annotation name*`)` right in the code. That is, to annotate the call to `gor()` in

```
z = bul() + gor(42);
```

as `spectime`, you would write

```
z = bul() + __CMIX(spectime)gor(42);
```

If you use this syntax you'll probably also want to insert something like

```
#ifndef __CMIX
#  define __CMIX(garbage) /*nothing*/
#endif
```

at the top of the affected files, so your progam will also compile without C-Mix. Or maybe even (though it might interfere with our ability to write specializer directives as `#pragma`s on systems where such lines are subject to macro replacement):

```
#ifdef __CMIX
#  define spectime __CMIX(spectime)
#else
#  define spectime /*nothing*/
#endif
/* blah blah */
z = bul() + spectime gor(42);
```

4.5 Commentary on the `dangerous spectime` annotation

The problem with `dangerous spectime` is that C-Mix/II cannot take the value of a such-annotated variable into account when it considers whether an already generated piece of residual code can be reused in a given situation. Consider this code:

```
extern int ex ;
int in ;
#pragma cmix spectime: in ; dangerous spectime: ex ;
void showthem() {
    __CMIX(spectime) printf("ex=%d, in=%d\n",ex,in);
}
#pragma cmix generator: main specializes goal()
void goal() {
    ex = 117 ; in = 1001 ;
    showthem();
    in = 42 ;
    showthem();
    ex = 1984 ;
    showthem();
}
```

This program causes only *two* residual versions of `showthem()` to be generated: one for `in` $= 1001$ and one for `in` $= 42$, and each one will have embedded the value `ex` has when it is first discovered that the function is needed. This happens to be 117, so the residual program will (errorneously) output,

```
ex=117, in=1001
ex=117, in=42
ex=117, in=42
```

And worse yet: C-Mix/II does not detect that there is any problem at all.

It follows that one has to be extremely careful when using this annotation. It is always safer and nearly always better to move the definition of the variable into C-Mix/II's view (possibly by moving it to a source file of its own) and use `visible spectime`.

The reason the annotation exists at all is that in a few cases the deficiency can be turned into a advantage. If the variable is *e.g.* a large array and you are really absolutely sure that C-Mix/II doesn't need to keep track of changes to it (ideally because there aren't any), then this annotation may cause the generating extension to use significantly less time and space. This is because then it doesn't need to remember the variable's previous contents and compare it every time code is to be shared.

5 Specializer directives

Specializer directives provide overall control on what C-Mix/II does. They define which C files are read, which function is to be specialized with which parameter binding times, etc. User annotations of the subject program are also given as specializer directives.

There are several ways to supply C-Mix/II with specializer directives:

- By `-e` switches on the command line. (Think sed, grep, perl).
- Implicitly on the command line: A command line argument that looks like a filename and ends in `.c` is converted to the specializer directive

  `source:` *whatever*`.c`
- In a separate *script file*. By convention script files have a `.cmx` suffix and the same name as the generating extension that should be produced. Any command line argument to C-Mix$_{\text{II}}$ that does not have any other meaning is interpreted as a script file name. The contents of the file is parsed as specializer directives, separated by semicolons or blank lines.
- As "pragma"s in the C sources read by C-Mix$_{\text{II}}$. If any line in the C input begins with

  ```
  #pragma cmix
  ```

  the rest of that line is parsed as specializer directive(s). The `source:` and `define:` directives cannot be given as pragmas.

  It varies from system to system whether preprocessor macros are expanded on `#pragma` lines. The differences do not come from C-Mix$_{\text{II}}$ but from the external preprocessor that was found when C-Mix$_{\text{II}}$ was installed.

The only directives that *need* to be present are a `source:` and a `goal:` or `generator:` directive.

5.1 The `goal:` directive

```
goal: foo(?,$1,?)
```

You use this directive to specify which form of residual functions you want the generating extension to output.

`foo` is the function in the subject program you want to create specialized versions of. In addition to its name you also need to specify the binding time of its parameters. For each parameter you say either

`?` – meaning that the parameter is residual. The residual function will have one parameter for each `?` in the directive; their order will be the same as they have in the subject program.

`$`n – meaning that the parameter's value will be given at spectime as command line argument n to the generating extension.

When you use the `goal:` directive to specify the goal function, the generating extension produced by C-Mix$_{\text{II}}$ will be (the C source for) a stand-alone program. The generating extension reads the "`$`n" parameters from its command line (if any "`$`n" specifier does not occur in the parameter specification—*e.g.* , `specializes foo(?,$2)`—the command line parameters with "unused" numbers will simply be ignored). It then generates a specialized function definition and writes it to the standard output stream. The specialized function will have the same name as the original one, but it will have fewer parameters, simply leaving the spectime ones out of the parameter list.

(If you want to select another name for the specialized function, you can attach a "`producing ...`" modifier to the `goal:` directive the same way as it is used in the `generator:` directive, which we'll explain in a while).

Examples. Assume that

```
int foo(int a,int b,struct bar c) { ... }
```

is in one of the source files. Then

```
goal: foo(?,$1,?)
```

tells C-Mix/II to produce a generating extension that excepts a single command line argument which it parses as the integer `b`. The generating extension outputs a residual program that includes an `int foo(int a,struct bar c)` that is specialized to the given `b`.

```
goal: foo(?,$1,$2)
```

is an error—the generating extension can't parse a command line argument as a `struct bar`. In general, only arithmetic types and pointer-to-char is accepted as spectime parameters.

There can only be one `goal:` directive in each C-Mix/II run. If you think you need more, read ahead and see the `generator:` directive.

5.2 The `generator:` directive

```
generator: quux specializes foo(?,$1,?)
   producing (zor%d,$1)
```

This is the `goal:` directive's older brother, used when you need more sophisticated control of the generating extension's operation.

When you use the `generator:` directive, the generating extension does *not* include a `main()` function—you must supply a main program yourself.

Your main program can obtain the spectime data in any way it wants to. When it is ready, it initializes the specialization module by calling

```
cmixGenInit()
```

and then calls the *generator entry point*

```
quux(spectime arguments)
```

with the spectime arguments. Finally it calls

```
cmixGenExit(file)
```

with an argument of type `FILE*`, which outputs the generated function definitions to the specified file.

Before the call to `cmixGenExit` you can set the global `int` variable `cmixRestruct` (which is exported from `libcmix.a`) to 0, which turns off a restructuring phase that tries to express the residual program with structured control-flow constructions instead of the simpe `goto`s C-Mix/II works with internally. Turning this phase off makes the call to `cmixGenExit` somewhat faster, but it may also cause the residual program to run slower with some compilers. We have also seen examples of the reverse: that a bare `goto`-based residual program runs faster that a well-structured version of the same program. If the last few percents of efficiency matters to you, experiment!

The parameters to the generator entry point naturally corresponds to "$\$n$" specifications in the directive just as do command line arguments when you use the `goal:` directive. But the restrictions on their possible types are much lessened; your main program can pass the generating extension structs and pointers and other types that do not have a straightforward ASCII syntax.

You can call the generator entry point multiple times between the calls to the `cmixGenInit` and `cmixGenExit` functions. That way you can, in a single run, generate multiple specialized versions of the same function, with the potential of sharing residual versions of helper functions. Or you can even generate specialized versions of several different functions in a single run. Just pick different entry point names and supply more than one `generator:` directive to C-Mix/II.

The potential for generating more than one function at a time means that you need control over which names the residual versions get. A file with 12 different function definitions all by the name of `foo` tends to confuse C compilers (and the programmer who wants to call one of the 12).

The residual name control is the task of the optional clause "`producing (zor%d,$1)`" in the example above. `zor%d` is a printf-like format string, and is followed by as many "$\$n$" argument as there are conversion specifiers in the format string. **Warning:** C-Mix/II does not attmpt to check the format string against the "$\$n$" arguments; it not even checks to see if their number is correct. They are simply copied directly to a `sprintf` call in the generating extension.

Examples. Assume (as in the section on `goal:`) that

```
int foo(int a,int b,struct bar c) { ... }
```

is in one of the source files.

```
generator: baz specializes foo(?,$1,$2)
```

makes a generator module that exports the generator function `void baz(int b, struct bar c)`. The generator function generates a residual `int foo(int a)`. Note that `foo(?,$1,$2)` was not allowed in a `goal:` directive because a stand-alone generating extension wouldn't know how to interpret a command-line argumen as a `struct bar`. This is not a problem when C-Mix/II doesn't have to invent a `main()` itself, though.

```
generator: baz specializes foo(?,$1,$2)
   producing(zor%d,$1)
```

specifies the same generator heading, but where the residual function is called *e.g.* `zor42` if the spectime `b` was 42.

```
generator: baz specializes foo(?,$1,$2) producing(%s,$3)
```

makes the generator function take a third parameter that gives the name of the residual entry point.

```
generator: baz specializes foo(?,$2,$1)
```

makes the generator be declared `void baz(struct bar c,int b)`.

```
generator: baz specializes foo($1,$1,?)
```

make a generator with heading `void baz(int ab)` that specializes `foo()` to the same number as first and second argument.

There may be more than one `generator:` directive—but there cannot be any if there is a `goal:` directive. Beware, though, that C-Mix/II will decide a single set of binding times for the program's variables that accomodate all of the requested generators. So if you say

```
generator: g1 specializes foo($1,$2,?) producing (%s,$3)
generator: g2 specializes foo(?,?,$1) producing (%s,$2)
```

you are forcing all three arguments to `foo` to be residual, which may not give you much speedup.

5.3 The `source:` directive

```
source: file1.c ../somewhere/else/file2.c
```

This directive names the C source files that contain the subject program.

More than one file name can be given, possibly in different directives. All of the specified files are treated as a single program to be specialized, but C's scope and linkage rules are observed (*i.e.* an identifier declared `static` in one file will be invisible in every other file). It is assumed that variables and functions defined with external linkage are shared between the files but not used outside the set of files that C-Mix$_{II}$ sees. (See section 4.1 for how to override this assumption for variables).

5.4 The user annotation directives

```
spectime: some_var some_fun()
residual: some_fun::cnt
visible spectime: global_var
visible residual: another_global yetone
pure spectime: bar()
pure: foo() capacity()
```

These have their own section in the manual, section 4.

5.5 The `define:` directive

```
define: BASEDIR=/home/myself/cmix/ DEBUG
```

is equivalent to inserting

```
#define BASEDIR "/home/myself/cmix/"
#define DEBUG
```

at the top of each C source file.

In addition to defining the symbols specified in specializer directives, C-Mix$_{II}$ passes any `-D`*whatever* options from its command line on to the C preprocessor.

C-Mix$_{II}$ always defines the symbol `__CMIX`. You can use `#ifdef __CMIX` etc. to make the C source that C-Mix$_{II}$ sees differ from the source that your C compiler sees if you compile your program without specializing it.

C-Mix$_{II}$ always defines the symbol `__STDC__` which indicates that it tries to be a standards-conformant C implementation.

> Is it wise to *always* do this? The standard specifies that a lot of the standard header constants should be "suitable for use in an #if". Perhaps we should only define `__STDC__` if we include headers supplied by the C compiler.

You can give any number of `define` directives.

5.6 The `outputbase:` directive

```
outputbase: output/my-project
```

This directive can be used to specify where C-Mix/II puts its output files. With the example above, the file containing the annotated program will be named `output/my-project.ann` and the C source for the generating extention will be written to a file called `output/my-project-gen.c`.

5.7 Other directives

```
signed chars are glyphs
```

This is a magic spell (must be typed exactly as given here) that changes the way C-Mix/II writes values of type `unsigned char` as constants in the residual program when they result from spectime computations. Normally C-Mix/II assumes that when the programmer specifies `unsigned char` he really means "very short int"; thus they are written to the residual program as integral constants.

Sometimes, however, people use `unsigned char` for storing characters because they want to be 8-bit clean and the plain `char` type in their compiler is signed. With that kind of programs, this directive can make the residual programs more readable and increase the probability that it can be ported to another system that uses another character set and still work as expected[6].

```
lift long double
```

This magic spell causes the generating extension to use more significant digits when writing intermediate results of type `long double` (or abstract numeric or floating-point type) into the text of the residual program. Normally such values get truncated to `double` because the `printf` routine used by C-Mix/II cannot handle `long double` correctly on some systems.

6 Writing abstract header files

Using abstract interfaces is a well-known technique in software development. In C it is typically employed by definig the abstract interface in a heder file. It is not uncommon that the header file needs, for the benefit of the compiler, to include details about the implementation of the interface. One example is the `FILE` type defined by `stdio.h` or the constants defined in `limits.h`.

One use for a partial evaluator such as C-Mix/II is to specialize the code that uses the abstract interface with respect to the implementation of it. In some cases this can lead to considerable speedup, because partial evaluation can eliminitate the run-time negotiation between the user code and the (otherwise separately compiled) implementation. Specializing abstract interfaces away in this fasion is not a problem with C-Mix/II. You just let C-Mix/II read the user source and the implementation source files; C-Mix/II automatically invokes a C preprocessor that interpolates the contents of the header files, and voila—the code gets specialized with respect the implementation details.

[6] Note that there are numerous other issues that can cause residual programs to fail when run on other machines than the one where the generating extension ran on. The C standard allows programs to depend in obscure ways on the target machine's way of doing arithmetic and the widths of built-in types, and C-Mix/II makes no attempts to guarantee correctness of cross-specialization or even warn its user when that is the case.

However, it happens just as well that this is precisely *not* what you wants. That can have several reasons:

- You might want to be able to change the implementation of the abstract code without changing the residual program. That means that the residual program must honor the same abstraction as the original source.

- You may not have access to the source code for the implementation of the abstract interface, only the partial information that have to go in the headers. That may not be enough to gain any useful specialization in itself so, *e.g.* , the expansion of preprocessor macros may have as its only effects putting a bigger load on the C-Mix$_{II}$ system and making the annotated program produced by the analyzer harder to read.

- The implementation may depend on non-standard syntactic extensions of the C compiler. Thus C-Mix$_{II}$ may not be able to read the input at all if the header files are interpolated normally.

The solution in this case is to create an *abstract header file* that is included instead of the real header file when the source is processed by C-Mix$_{II}$. This is the technique we use to support standard headers such as `stdio.h`. This section describes the constructions C-Mix$_{II}$ supports for introducing abstraction into its conception of the C language.

6.1 The abstract header file itself

There are two basic techniques for introducing abstractions.

1. Use conditional compilation directives to make the same header file come out differently when read by C-Mix$_{II}$ or the C compiler. The header file might look something like

   ```
   #ifndef MYHEADER_INCLUDED
     #define MYHEADER_INCLUDED
     #ifdef __CMIX
        /* define abstract interface here */
     #else
        /* define concrete interface here */
     #endif
   #endif
   ```

   This technique requires you to change the original source but may be the only option if the header file resides in the same directory as the `.c` files.

2. Write the abstract definition in a separate file with the same name as the original header file, and put it in a special directory. Use the `-I` option to make C-Mix$_{II}$ search special directory. This is often the cleanest solution and the one C-Mix$_{II}$ itself uses for the standard headers (n which case the right `-I` is implied).

An abstract header definition generally consists of

- Administrative specializer directives to make the residual code and the generating extension interface properly to the concrete definition of the interface at compile time. The `header`, `well-known` and `taboo` directives described below are used for this.

- Magic constant declarations with the `well-known constant` directive, discussed in section 6.3.

- Abstract type declarations. There is a special syntax for these, to be discussed in section 6.4.

- Normal declarations for function prototypes (possibly including "functions" which in the real header files are really function-like preprocessor macros) and/or external variables.

- User annotations for those function and variables (see section 4).

Many of these tasks are done with specializer directives. The `#pragma cmix` syntax for introducing specializer directives from C source code was motivated partly from the need to include specializer directives in abstract headers.

6.2 Support directives for abstract headers

6.2.1 The `header:` directive

```
header: myheader.h
```

This directive causes a

```
#include myheader.h
```

line to be emitted to the beginning of the generating extension source and the residual program source. C-Mix/II does not try to read `myheader.h` and does not even check that it is a valid filename.

6.2.2 The `well-known:` directive

```
well-known: thisfun thatvar
```

This directive prevents C-Mix/II from emitting declarations for external variables or functions named `thisfun` or `thatvar` when it emits the generating extension and the residual program source. The rest of the code is still generated as though a declaration was present, so you'll have to make sure by some other means you won't run into trouble with undeclared identifiers.

Note that function names do *not* take a `()` here as the do in user annotation directives.

The intended use is for external functions that get defined in headers included with `header:` directives. The "function" may actually be defined as a macro or have a slightly different prototype in the header than C-Mix/II thinks it should have. By instructing C-Mix/II to omit a declaration of its own, you can avoid confusing the compiler.

6.2.3 The `taboo:` directive

```
taboo: FLUSH INIT FIND PASS
```

This directive instructs C-Mix$_{II}$ to avoid the given identifiers when it chooses names for local functions and variables in the generating extension and residual program.

This directive is useful if a header file may define macros or additional functions that you do not want C-Mix$_{II}$ to know about.

The names of external functions and variables and `visible` variables are not affected by the `taboo:` directive. They come out exactly as they appear in C-Mix$_{II}$'s input, and if that creates problems, you get to sort out some interesting error messages from the C compiler. The same holds for `generator:` entry points.

6.3 Magic constants

Some abstract interfaces defines identifiers that should evaluate to values with a given type without defining what the precise value is. One example is `stdin`, `stdout`, and `stderr` defined by `<stdio.h>` to be expressions of type `FILE*` with specified properties.

One way to write an abstract definition for such an identifier would be

```
extern FILE *const stderr;
#pragma cmix well-known: stderr
```

which indeed makes `stderr` be a (pseudo-)rvalue expression of type `FILE*` and suppresses the declaration in the residual program.

However, there is an unexpected side effect of this. It makes C-Mix$_{II}$ decide on a single binding for the variable. Thus one cannot have `stderr` appear at spectime as well as in the residual program. This would have made good sense, e.g.

```
__CMIX(spectime)fprintf(stderr,p-gen went wrong\n);
__CMIX(residual)fprintf(stderr,p-res went wrong\n);
```

This use is made possible with the `well-known constant` annotation:

```
well-known constant: foo
```

Whenever `foo` appears in the C source it is treated as a literal constant just like, *e.g.* , "42", and copied verbatim to the generating extension or the residual program (the binding-time analysis decides which). It is assumed that a header will be included (with the `header:` directive) that makes the precense of the identifer there meaningful.

`foo` must be declared somewhere in the program as an `extern` and `const` variable—this declaration supplies the type of the constant expression and must be in scope whenever it is used[7]. The declaration is automatically supressed in the output, and `foo` is implicitly registered as a `taboo`.

6.4 Abstract types

C-Mix$_{II}$ has a special syntax for defining abstract types:

[7]The reason for this convoluted two-step syntax is that it frees the specializer directive parser from containing a parser for C types

```
typedef __CMIX(bar) baz ;
```

which declares `baz` as an abstract type with the `b`, `a`, and `r` properties. The type so defined is treated very like a predefined type such as `signed char`. Syntactically the abstract type is a "typedef-name".

Zero or more of the following property characters can be specified between the parentheses:

- `a` – The type is "arithmetic": the type checker will allow values of this type as operands to arithmetic operators such as `+` and `/`, and conversions to and from other arithmetic types.

- `i` – The type is "integral": the type checker allows the same operations as for arithmetic types, in addition to integer-only operations such as `%` and `<<`.

- `u` – The type is "unsigned": This is only meaningful if the `i` property is given. If lifting of the type is required, the value to be lifted will be converted to an `unsigned long` and printed to the residual program as such.

- `s` – The type is "signed". This counterpart to `u` is currently the default for "integral" abstract types with no signedness attribute. When lifting the type it gets printed as a `long`.

- `p` – The type is a "pointer type": the type checker will allow conversions to and from other pointer types.

Unrecognized properties such as `b` and `r` are silently ignored.

It is expected that `header` directives are used to include headers that make it legal to use `foo` as a type name in the generating extension and the residual program.

7 The past and future of C-Mix/II

7.1 Historic background

The original C-Mix was written by Lars Ole Andersen in 1991 as a prototype implementation of the theory in his master's thesis. The system evolved over the years, maintained by Lars Ole Andersen and Peter Holst Andersen, who both continued to develop theory and techniques—after C specialization had earned Lars Ole his Ph.D. and he left DIKU, Peter Holst Andersen kept extending the system.

But the old C-Mix was primarily a research prototype and its user interface did not have top priority. The system never became very easy to use, and it had a disturbing tendency to revenge bitterly on its user when faced with a construct it couldn't make sense of (whether because its input was actually illegal C or the feature had just not been implemented properly).

In early 1997 we began a project of rewriting C-Mix from scratch with emphasis on stabiltiy and an easy-to-use interface. This effort concluded late in February 1999 with the release of C-Mix 2.0.0.

7.2 Future improvements

These are the improvements to C-Mix$_{II}$ we're planning to make in later releases:

- **Utilizing liveness information** to reduce the number of locations that are considered when comparing spectime states. This will improve the running times of the generating extension, in addition to reducing code blowup in the residual program.
- **The ability to "lift" strings** so they can be computed at spectime and used in the residual program. This is not always safe, and we need to develop analyses or annotations to determine when.
- **Structures with mixed binding time**. We already have most of the machinery to support structures where some members are spectime and others are residual. This can be implemented with rather little extra effort.
- **Better, more coherent documentation**
- **Beautifying post-processing of the residual program**. The current C-Mix$_{II}$ outputs a residual program in a very low-level programming style: a lot of very simple statements, and `goto`s as the primary control structure.

  If we could implement heuristics for expressing the control structure with Cs looping and branching primitives, it would make the residual programs much more readable for human readers. That would make it easier to verify that one has got the expected result from partial evaluation, and it would make C-Mix$_{II}$ more useful as a general introduction to partial evaluation.

  Additionally it turns out—rather to our surprise, actually—that several C compilers actually generate better machine code from a `for` loop than from the same loop when it has been translated into `goto`s.

  Similar considerations can be made about the very simple statements that make up the computations in the residual program. It would be nice to be able to combine

  ```
  iTmp = f(42); a = iTmp + b;
  ```

  into

  ```
  a = f(42)+b;
  ```

- **Polyvariant specialization**. The ability to specialize a function, not only to different spectime values, but to totally different binding times, would make it a lot easier to get good performance out of specialized program without modifying the subject program's text too much. As one can imagine this is going to require substantial changes to C-Mix$_{II}$'s internals, so don't hold your breath.

References

[SRGP 1990] Simple Raster Graphics Package *from Computer Graphics: Principles and Practice* by Foley, vanDam, Feiner and Hughes. Available from http://www.cs.brown.edu/software/catalog.html

[PTC 1998] Prometheus Truecolour 2.0. A graphics API available from http://www.gaffer.org/ptc/

[OpenGL 1999] OpenGL. A graphics API available from http://www.sgi.com/software/opengl/

[Mesa 1999] Mesa. The free OpenGL work-alike graphics API available from http://www.mesa3d.org/